Capítulo VINTE E DOIS
Fusos Horários e Horário de Verão

Objetivos do Exame
Trabalhar com datas e horários em diferentes fusos horários e gerenciar mudanças resultantes do horário de verão, incluindo a formatação de valores de data e hora.

Classes centrais de fuso horário

Antes do Java 8, se quiséssemos trabalhar com informações de fuso horário, tínhamos que usar a classe java.util.TimeZone. Agora, com a nova API de Data/Hora, existem opções novas e melhores.

Elas são:

- **ZoneID**
  Representa o ID do fuso horário. Por exemplo, Asia/Tokyo.
- **ZoneOffset**
  Representa um deslocamento de fuso horário. É uma subclasse de ZoneId. Por exemplo, -06:00.
- **ZonedDateTime**
  Representa uma data/hora com informações de fuso horário. Por exemplo, 2015-08-30T20:05:12.463-05:00[America/Mexico_City].
- **OffsetDateTime**
  Representa uma data/hora com um deslocamento de UTC/Greenwich. Por exemplo, 2015-08-30T20:05:12.463-05:00.
- **OffsetTime**
  Representa uma hora com um deslocamento de UTC/Greenwich. Por exemplo, 20:05:12.463-05:00.

Assim como as classes do capítulo anterior, essas estão localizadas no pacote java.time e são imutáveis.

(continua na próxima resposta)

Classes ZoneId e ZoneOffset

O mundo está dividido em fusos horários nos quais o mesmo horário padrão é mantido. Por convenção, um fuso horário é expresso como o número de horas de diferença em relação ao Tempo Universal Coordenado (UTC). Como o Horário Médio de Greenwich (GMT) e o Tempo Zulu (Z), usado no meio militar, não têm deslocamento em relação ao UTC, eles são frequentemente usados como sinônimos.

O Java usa o banco de dados de fusos horários da **IANA** (Internet Assigned Numbers Authority), que mantém um registro de todos os fusos horários conhecidos ao redor do mundo e é atualizado muitas vezes por ano.

Cada fuso horário tem um ID, representado pela classe java.time.ZoneId. Existem três tipos de ID:

1. O primeiro tipo apenas declara o deslocamento em relação ao tempo UTC/GMT. Eles são representados pela classe ZoneOffset e consistem em dígitos começando com + ou -, por exemplo, +02:00.
2. O segundo tipo também declara o deslocamento em relação ao tempo UTC/GMT, mas com um dos seguintes prefixos: UTC, GMT e UT, por exemplo, UTC+11:00. Eles também são representados pela classe ZoneOffset.
3. O terceiro tipo é baseado em região. Esses IDs têm o formato área/cidade, por exemplo, Europe/London.

Você pode obter todos os IDs de zona disponíveis com o método estático:

```java
Set<String> getAvailableZoneIds()
```

Por exemplo, para imprimi-los no console:

```java
ZoneId.getAvailableZoneIds().stream().forEach(System.out::println);
```

Para obter o ID de zona do seu sistema, use o método estático:

```java
ZoneId.systemDefault()
```

Internamente, ele usa java.util.TimeZone.getDefault() para encontrar o fuso horário padrão e o converte para um ZoneId.

Se você quiser criar um objeto ZoneId específico, use o método of():

```java
ZoneId singaporeZoneId = ZoneId.of("Asia/Singapore");
```

Esse método analisa o ID produzindo um ZoneOffset ou um ZoneRegion (ambos estendem ZoneId).

Na verdade, a linha acima produz um ZoneRegion. Um ZoneOffset é retornado se, por exemplo, o ID for Z, ou começar com + ou -. Por exemplo:

```java
ZoneId zoneId = ZoneId.of("Z"); // Z representa o ID de zona para UTC
ZoneId zoneId = ZoneId.of("-2"); // -02:00
```

As regras para esse método são:

- Se o ID de zona for igual a Z, o resultado é ZoneOffset.UTC. Qualquer outra letra lançará uma exceção.
- Se o ID de zona começar com + ou -, o ID será interpretado como um ZoneOffset usando ZoneOffset.of(String).
- Se o ID de zona for igual a GMT, UTC ou UT, então o resultado é um ZoneId com o mesmo ID e regras equivalentes a ZoneOffset.UTC.
- Se o ID de zona começar com UTC+, UTC-, GMT+, GMT-, UT+ ou UT-, então o ID é dividido em dois, com um prefixo de duas ou três letras e um sufixo começando com o sinal. O sufixo é analisado como um ZoneOffset. O resultado será um ZoneId com o prefixo especificado e o ID de deslocamento normalizado.
- Todos os outros IDs são interpretados como IDs de zona baseados em região. Se o formato for inválido (ele deve coincidir com a expressão [A-Za-z][A-Za-z0-9~/._+-]+) ou não for encontrado, uma exceção será lançada.

Lembre-se de que um ZoneOffset representa um deslocamento, geralmente em relação ao UTC. Esta classe possui muito mais construtores do que ZoneId:

```java
// O deslocamento deve estar no intervalo de -18 a +18
ZoneOffset offsetHours = ZoneOffset.ofHours(1);

// O intervalo é de -18 a +18 para horas e de 0 a ±59 para minutos
// Se as horas forem negativas, os minutos devem ser negativos ou zero
ZoneOffset offsetHrMin = ZoneOffset.ofHoursMinutes(1, 30);

// O intervalo é de -18 a +18 para horas e de 0 a ±59 para minutos e segundos
// Se as horas forem negativas, minutos e segundos devem ser negativos ou zero
ZoneOffset offsetHrMinSe = ZoneOffset.ofHoursMinutesSeconds(1,30,0);

// O deslocamento deve estar no intervalo -18:00 a +18:00
// O que corresponde a -64800 a +64800
ZoneOffset offsetTotalSeconds = ZoneOffset.ofTotalSeconds(3600);

// O intervalo deve estar entre +18:00 e -18:00
ZoneOffset offset = ZoneOffset.of("+01:30:00");
```

Os formatos aceitos pelo método of() são:

```diff
Z (para UTC)
+h
+hh
+hh:mm
-hh:mm
+hhmm
-hhmm
+hh:mm:ss
-hh:mm:ss
+hhmmss
-hhmmss
```

Se você passar um formato inválido ou um valor fora do intervalo para qualquer um desses métodos, uma exceção será lançada.

Para obter o valor do deslocamento, você pode usar:

```java
// Obtém o deslocamento como int
int offsetInt = offset.get(ChronoField.OFFSET_SECONDS);

// Obtém o deslocamento como long
long offsetLong = offset.getLong(ChronoField.OFFSET_SECONDS);

// Obtém o deslocamento em segundos
int offsetSeconds = offset.getTotalSeconds();
```

ChronoField.OFFSET_SECONDS é o único valor aceito de ChronoField, então as três instruções acima retornam o mesmo resultado. Outros valores lançam uma exceção.

De qualquer forma, uma vez que você tenha um objeto ZoneId, pode usá-lo para criar uma instância de ZonedDateTime.

Um objeto ZonedDateTime

```
text                                                    Copiar  Editar

        Data                                Deslocamento
2015-08-31 T08:45:20.000 +02:00[Africa/Cairo]

                            Hora                            Fuso horário
```

Classe ZonedDateTime

Um objeto java.time.ZonedDateTime representa um ponto no tempo relativo a um fuso horário.

Um objeto ZonedDateTime tem três partes:

- Uma data
- Uma hora
- Um fuso horário

Isso significa que ele armazena todos os campos de data e hora, com precisão de nanossegundos, e um fuso horário com um deslocamento de zona.

Assim, uma vez que você tenha um objeto ZoneId, pode combiná-lo com um LocalDate, LocalDateTime ou Instant, para transformá-lo em um ZonedDateTime:

```java
ZoneId australiaZone = ZoneId.of("Australia/Victoria");

LocalDate date = LocalDate.of(2010, 7, 3);
ZonedDateTime zonedDate = date.atStartOfDay(australiaZone);

LocalDateTime dateTime = LocalDateTime.of(2010, 7, 3, 9, 0);
ZonedDateTime zonedDateTime = dateTime.atZone(australiaZone);

Instant instant = Instant.now();
ZonedDateTime zonedInstant = instant.atZone(australiaZone);
```

Ou usando o método of:

```java
ZonedDateTime zonedDateTime2 =
    ZonedDateTime.of(LocalDate.now(), LocalTime.now(), australiaZone);
ZonedDateTime zonedDateTime3 =
    ZonedDateTime.of(LocalDateTime.now(), australiaZone);
ZonedDateTime zonedDateTime4 =
    ZonedDateTime.ofInstant(Instant.now(), australiaZone);
// ano, mês, dia, hora, minutos, segundos, nanossegundos, zoneId
ZonedDateTime zonedDateTime5 =
    ZonedDateTime.of(2015,1,30,13,59,59,999, australiaZone);
```

Você também pode obter a data/hora atual do relógio do sistema no fuso horário padrão com:

```java
ZonedDateTime now = ZonedDateTime.now();
```

A partir de um ZonedDateTime, você pode obter LocalDate, LocalTime ou LocalDateTime (sem a parte do fuso horário) com:

```java
LocalDateTime currentDate = now.toLocalDate();
LocalDateTime currentTime = now.toLocalTime();
LocalDateTime currentDateTime = now.toLocalDateTime();
```

ZonedDateTime também possui a maioria dos métodos de LocalDateTime que vimos no capítulo anterior:

```java
// Para obter o valor de um campo específico
int day = now.getDayOfMonth();
int dayYear = now.getDayOfYear();
int nanos = now.getNano();
Month monthEnum = now.getMonth();
int year = now.get(ChronoField.YEAR);
long micro = now.getLong(ChronoField.MICRO_OF_DAY);

// Isso é novo, obtém o deslocamento da zona como "-03:00"
ZoneOffset offset = now.getOffset();

// Para criar outra instância
ZonedDateTime zdt1 = now.with(ChronoField.HOUR_OF_DAY, 10);
ZonedDateTime zdt2 = now.withSecond(49);

// Como esses métodos retornam uma nova instância, podemos encadeá-los!
ZonedDateTime zdt3 = now.withYear(2013).withMonth(12);
```

Os dois métodos a seguir são específicos de ZonedDateTime:

```java
// Retorna uma cópia da data/hora com um
// fuso horário diferente, mantendo o instante
ZonedDateTime zdt4 = now.withZoneSameInstant(australiaZone);

// Retorna uma cópia desta data/hora com um fuso horário diferente,
// mantendo a data/hora local, se for válida para o novo fuso horário
ZonedDateTime zdt5 = now.withZoneSameLocal(australiaZone);
```

Adição

```java
ZonedDateTime zdt6 = now.plusDays(4);
ZonedDateTime zdt7 = now.plusWeeks(3);
ZonedDateTime zdt8 = newYear2001.plus(2, ChronoUnit.HOURS);
```

Subtração

```java
ZonedDateTime zdt9 = now.minusMinutes(20);
ZonedDateTime zdt10 = now.minusNanos(99999);
ZonedDateTime zdt11 = now.minus(10, ChronoUnit.SECONDS);
```

O método toString() retorna a data/hora no formato de um LocalDateTime seguido por um ZoneOffset, opcionalmente um ZoneId se ele for diferente do offset, e omite as partes com valor zero:

```java
// Imprime 2014-09-19T00:30Z
System.out.println(
    ZonedDateTime.of(2014,9,19,0,30,0,0,ZoneId.of("Z")));

// Imprime 2015-08-31T12:39:27.492-04:00[America/Montreal]
System.out.println(
    ZonedDateTime.now(ZoneId.of("America/Montreal")));
```

Horário de verão

Muitos países no mundo adotam o que é chamado de Horário de Verão (Daylight Saving Time - DST), a prática de adiantar o relógio em uma hora no verão (bem, não exatamente no verão em todos os países, mas vamos seguir com isso) quando o horário de verão começa.

Quando o horário de verão termina, os relógios são atrasados em uma hora. Isso é feito para fazer melhor uso da luz natural do dia.

ZonedDateTime está totalmente ciente do horário de verão (DST).

Por exemplo, vamos considerar um país onde o DST é totalmente observado, como a Itália (UTC/GMT +2).

Em 2015, o DST começou na Itália em 29 de março e terminou em 25 de outubro. Isso significa que em:

- 29 de março de 2015 às 02:00:00 da manhã os relógios foram adiantados 1 hora para 29 de março de 2015 às 03:00:00 da manhã, horário local de verão.
  (Portanto, um horário como 29 de março de 2015 às 02:30:00 da manhã realmente não existiu!)
- 25 de outubro de 2015 às 03:00:00 da manhã os relógios foram atrasados 1 hora para 25 de outubro de 2015 às 02:00:00 da manhã, horário local de verão.
  (Portanto, um horário como 25 de outubro de 2015 às 02:30:00 da manhã existiu duas vezes!)

Se criarmos uma instância de LocalDateTime com essa data/hora e a imprimirmos:

```java
LocalDateTime ldt = LocalDateTime.of(2015, 3, 29, 2, 30);
System.out.println(ldt);
```

O resultado será:

```makefile
2015-03-29T02:30
```

Mas se criarmos uma instância de ZonedDateTime para a Itália (note que o formato usa uma cidade, não um país) e imprimirmos:

```java
ZonedDateTime zdt = ZonedDateTime.of(
     2015, 3, 29, 2, 30, 0, 0, ZoneId.of("Europe/Rome"));
System.out.println(zdt);
```

O resultado será exatamente como no mundo real ao usar DST:

```makefile
2015-03-29T03:30+02:00[Europe/Rome]
```

Mas cuidado. Temos que usar um ZoneId regional — um ZoneOffset não fará o trabalho, pois essa classe não possui as informações de regras de zona necessárias para considerar o horário de verão:

```java
ZonedDateTime zdt1 = ZonedDateTime.of(
     2015, 3, 29, 2, 30, 0, 0, ZoneOffset.ofHours(2));
ZonedDateTime zdt2 = ZonedDateTime.of(
     2015, 3, 29, 2, 30, 0, 0, ZoneId.of("UTC+2"));
System.out.println(zdt1);
System.out.println(zdt2);
```

O resultado será:

```makefile
2015-03-29T02:30+02:00[UTC+02:00]
2015-03-29T02:30+02:00
```

Quando o DST termina, algo semelhante acontece:

```java
LocalDateTime ldt = LocalDateTime.of(2015, 10, 25, 3, 30);
System.out.println(ldt);
```

O resultado será:

```makefile
2015-10-25T03:30
```

Ao criarmos uma instância de ZonedDateTime para a Itália, devemos adicionar uma hora para ver o efeito (caso contrário, estaremos criando o ZonedDateTime às 03:00 do novo horário):

```java
ZonedDateTime zdt = ZonedDateTime.of(
    2015, 10, 25, 2, 30, 0, 0, ZoneId.of("Europe/Rome"));
ZonedDateTime zdt2 = zdt.plusHours(1);

System.out.println(zdt);
System.out.println(zdt2);
```

O resultado será:

```makefile
2015-10-25T02:30+02:00[Europe/Rome]
2015-10-25T02:30+01:00[Europe/Rome]
```

Também precisamos ter cuidado ao ajustar o horário através da fronteira do horário de verão usando uma versão dos métodos plus() e minus() que recebe uma implementação de TemporalAmount, em outras palavras, um Period ou uma Duration. Isso porque ambos diferem em seu tratamento do horário de verão.

Considere uma hora antes do início do DST na Itália:

```java
ZonedDateTime zdt = ZonedDateTime.of(
    2015, 3, 29, 1, 0, 0, 0, ZoneId.of("Europe/Rome"));
```

Quando adicionamos uma Duration de um dia:

```java
System.out.println(zdt.plus(Duration.ofDays(1)));
```

O resultado é:

```makefile
2015-03-30T02:00+02:00[Europe/Rome]
```

Quando adicionamos um Period de um dia:

```java
System.out.println(zdt.plus(Period.ofDays(1)));
```

O resultado é:

```makefile
2015-03-30T01:00+02:00[Europe/Rome]
```

O motivo é que Period adiciona um dia conceitual, enquanto Duration adiciona exatamente um dia (24 horas ou 86.400 segundos), e quando atravessa a fronteira do horário de verão, uma hora é adicionada, e o horário final é 02:00 em vez de 01:00.

OffsetDateTime e OffsetTime

OffsetDateTime representa um objeto com informações de data/hora e um deslocamento em relação ao UTC, por exemplo:

```makefile
2015-01-01T11:30-06:00
```

Você pode pensar que Instant, OffsetDateTime e ZonedDateTime são muito parecidos — afinal, todos armazenam data e hora com precisão de nanossegundos. No entanto, há diferenças sutis mas importantes:

- Instant representa um ponto no tempo no fuso horário UTC.
- OffsetDateTime representa um ponto no tempo com um deslocamento (qualquer deslocamento).
- ZonedDateTime representa um ponto no tempo em um fuso horário (qualquer fuso horário), acrescentando regras completas de fuso, como ajustes de horário de verão.

OffsetTime representa uma hora com um deslocamento do UTC, por exemplo, 11:30-06:00. A maneira comum de criar instâncias dessas classes é:

```java
OffsetDateTime odt = OffsetDateTime.of(
    LocalDateTime.now(), ZoneOffset.of("+03:00"));

OffsetTime ot = OffsetTime.of(
    LocalTime.now(), ZoneOffset.of("-08:00"));
```

Ambas as classes possuem praticamente os mesmos métodos que suas contrapartes LocalDateTime, ZonedDateTime e LocalTime. Com um deslocamento do UTC e sem variações de fuso horário, elas sempre representam um instante exato no tempo, o que pode ser mais adequado para certos tipos de aplicações (a documentação do Java recomenda OffsetDateTime ao se comunicar com banco de dados ou em protocolos de rede).

Análise e Formatação

java.time.format.DateTimeFormatter é a nova classe para analisar (parse) e formatar datas. Ela pode ser usada de duas formas:

As classes de data/hora LocalDate, LocalTime, LocalDateTime, ZonedDateTime, OffsetDate e OffsetDateTime todas possuem os seguintes três métodos:

```java
// Formata o objeto de data/hora usando o formatador especificado
String format(DateTimeFormatter formatter)

// Obtém uma instância de um objeto de data/hora (do tipo T)
// a partir de uma string com formato padrão
static T parse(CharSequence text)

// Obtém uma instância de um objeto de data/hora (do tipo T)
// a partir de uma string usando um formatador específico
static T parse(CharSequence text, DateTimeFormatter formatter)
```

DateTimeFormatter possui os seguintes dois métodos:

```java
// Formata um objeto de data/hora usando a instância do formatador
String format(TemporalAccessor temporal)

// Analisa o texto produzindo um objeto temporal
TemporalAccessor parse(CharSequence text)
```

Todos os métodos de formatação lançam a exceção em tempo de execução:

```java
java.time.DateTimeException
```

Todos os métodos de análise (parse) lançam a exceção em tempo de execução:

```java
java.time.format.DateTimeParseException
```

DateTimeFormatter fornece três maneiras de formatar objetos de data/hora:

- **Formatadores pré-definidos**
- **Formatadores específicos de localidade (locale)**

- **Formatadores com padrões personalizados**

Formatadores pré-definidos

Formatador	Descrição	Exemplo
BASIC_ISO_DATE	Campos de data sem separadores	20150803
ISO_LOCAL_DATE	Campos de data com separadores	2015-08-03
ISO_LOCAL_TIME	Campos de hora com separadores	13:40:10
ISO_LOCAL_DATE_TIME	Data e hora	2015-08-03T13:40:10
ISO_OFFSET_DATE	Data com deslocamento	2015-08-03+07:00
ISO_OFFSET_TIME	Hora com deslocamento	13:40:10+07:00
ISO_OFFSET_DATE_TIME	Data e hora com deslocamento	2015-08-03T13:40:10+07:00
ISO_ZONED_DATE_TIME	Data e hora com fuso	2015-08-03T13:40:10+07:00[Asia/Bangkok]
ISO_DATE, ISO_TIME	Data ou hora com ou sem deslocamento	2015-08-03+07:00, 13:40:10
ISO_DATE_TIME	Data/hora com ZoneId	2015-08-03T13:40:10+07:00[Asia/Bangkok]
ISO_INSTANT	Instante UTC	2015-08-03T13:40:10Z
ISO_ORDINAL_DATE	Ano e dia do ano	2015-200
ISO_WEEK_DATE	Ano, semana e dia da semana	2015-W34-2
RFC_1123_DATE_TIME	Formato de data RFC 1123 / RFC 822	Mon, 3 Ago 2015 13:40:10 GMT

Estilos Localizados

Você também pode usar estilos localizados para formatar datas e horários. Os estilos disponíveis são:

Estilo	Data	Hora
SHORT	8/3/15	1:40 PM
MEDIUM	Aug 03, 2015	1:40:00 PM
LONG	August 03, 2015	1:40:00 PM PDT
FULL	Monday, August 03, 2015	1:40:00 PM PDT

Símbolos de Padrões Personalizados

Símbolo	Significado	Exemplos
G	Era	AD; Anno Domini; A

Símbolo	Significado	Exemplos
u	Ano	2015; 15
y	Ano da era	2015; 15
D	Dia do ano	150
M / L	Mês do ano	7; 07; Jul; July; J
d	Dia do mês	20
Q / q	Trimestre do ano	2; 02; Q2; 2nd quarter
Y	Ano baseado em semana	2015; 15
w	Semana do ano (baseado em semana)	30
W	Semana do mês	2
E	Dia da semana	Tue; Tuesday; T
e / c	Dia da semana local	2; 02; Tue; Tuesday; T
F	Ocorrência do dia na semana no mês	2
a	AM/PM do dia	AM
h	Hora (1–12)	10
K	Hora (0–11)	1
k	Hora (1–24)	20
H	Hora (0–23)	23
m	Minuto	10
s	Segundo	11
S	Fração de segundo	999
A	Milissegundos do dia	2345
n	Nanossegundos do segundo	865437987
N	Nanossegundos do dia	12986497300
V	ID do fuso horário	Asia/Manila; Z; -06:00
z	Nome do fuso horário	Pacific Standard Time; PST
O	Deslocamento localizado	GMT+4; GMT+04:00; UTC-04:00
X	Deslocamento de zona ("Z" para zero)	Z; -08; -0830; -08:30

Símbolo	Significado	Exemplos
x	Deslocamento de zona	+0000; -08; -0830; -08:30
Z	Deslocamento de zona	+0000; -0800; -08:00
'	Escape para texto	'GMT' → GMT
''	Aspas simples	'' → '
[]	Seção opcional (início/fim)	
{}	Reservado para uso futuro	

Exemplos de Padrões Personalizados

Supondo:

```java
LocalDate ldt = LocalDate.of(2015, 1, 20);
```

Exemplos usando um formatador pré-definido:

```java
System.out.println(DateTimeFormatter.ISO_DATE.format(ldt));
System.out.println(ldt.format(DateTimeFormatter.ISO_DATE));
```

Saída:

```yaml
2015-01-20
2015-01-20
```

Exemplos usando um estilo localizado:

```java
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);
// Com o locale atual
System.out.println(formatter.format(ldt));
System.out.println(ldt.format(formatter));
// Com outro locale
System.out.println(formatter.withLocale(Locale.GERMAN).format(ldt));
```

Saída possível:

```swift
20/01/15
20/01/15
20.01.15
```

Exemplos com um padrão personalizado:

```java
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("QQQQ Y");
// Com o locale atual
System.out.println(formatter.format(ldt));
System.out.println(ldt.format(formatter));
// Com outro locale
System.out.println(formatter.withLocale(Locale.GERMAN).format(ldt));
```

Saída possível:

```yaml
1st quarter 2015
1st quarter 2015
1. Quartal 2015
```

Se o formatador usar informações que não estão disponíveis, uma DateTimeException será lançada. Por exemplo, ao usar DateTimeFormatter.ISO_OFFSET_DATE com uma instância de LocalDate (que não possui informações de deslocamento).

Análise (Parse) de valores de data/hora a partir de uma string

Para analisar um valor de data e/ou hora a partir de uma string, use um dos métodos parse(). Por exemplo:

```java
// Formato conforme ISO-8601
String dateTimeStr1 = "2015-06-29T14:45:30";

// Formato personalizado
String dateTimeStr2 = "2015/06/29 14:45:30";

// Análise com o formato padrão
LocalDateTime ldt = LocalDateTime.parse(dateTimeStr1);

// Usando um DateTimeFormatter
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

// DateTimeFormatter retorna uma instância de TemporalAccessor
TemporalAccessor ta = formatter.parse(dateTimeStr2);

// LocalDateTime retorna uma instância do mesmo tipo
ldt = LocalDateTime.parse(dateTimeStr2, formatter);
```

A versão de parse() das classes de data/hora aceita uma string em formato conforme a norma ISO-8601. Veja a tabela:

Classe	**Formato ISO-8601**	**Exemplo**
LocalDate	uuuu-MM-dd	2007-12-03
LocalTime	HH:mm:ss	10:15:30
LocalDateTime	uuuu-MM-dd'T'HH:mm:ss	2007-12-03T10:15:30
ZonedDateTime	uuuu-MM-dd'T'HH:mm:ssXXXXX[VV]	2011-12-03T10:15:30+01:00[Europe/Paris]
OffsetDateTime	uuuu-MM-dd'T'HH:mm:ssXXXXX	2011-12-03T10:15:30+01:00

Classe	Formato ISO-8601	Exemplo
OffsetTime	HH:mm:ssXXXXX	10:15:30+01:00

Se o formatador utilizar informações não disponíveis, ou se o padrão do formato for inválido, uma DateTimeParseException será lançada.

Pontos-Chave

- ZoneId, ZoneOffset, ZonedDateTime, OffsetDateTime e OffsetTime são as classes da nova API de Data/Hora do Java que armazenam informações sobre fusos horários e deslocamentos. Elas estão localizadas no pacote java.time e são imutáveis.
- Cada fuso horário possui um ID, representado pela classe ZoneId. Existem três tipos:
    1. Apenas deslocamento, como +02:00 (ZoneOffset)
    2. Deslocamento com prefixo UTC, GMT ou UT, como UTC+11:00 (ZoneOffset)
    3. Baseado em região, como Europe/London
- Para criar um objeto ZoneId específico, use:

```java
ZoneId.of("Asia/Singapore");
ZoneId.of("+3");
```

- Os dois primeiros produzem um ZoneRegion. O último, um ZoneOffset.
- Um objeto ZonedDateTime representa um ponto no tempo relativo a um fuso horário. Pode ser criado com:

```java
ZoneId australiaZone = ZoneId.of("Australia/Victoria");
ZonedDateTime zonedDateTime5 =
    ZonedDateTime.of(2015,1,30,13,59,59,999, australiaZone);
```

- Ou usando LocalDate, LocalTime, LocalDateTime ou Instant + ZoneId.
- ZonedDateTime respeita o horário de verão, ajustando a hora automaticamente quando ele começa ou termina.
- Period e Duration tratam o horário de verão de forma diferente:
    - Period adiciona um "dia conceitual"
    - Duration adiciona exatamente 24 horas
- OffsetDateTime representa um ponto no tempo com deslocamento em relação ao UTC, por exemplo, 2015-01-01T11:30-06:00.
- OffsetTime representa uma hora com deslocamento do UTC, como 11:30-06:00.
- DateTimeFormatter é a nova classe para formatação/análise de datas, usada com:
    - format(DateTimeFormatter)
    - parse(CharSequence)
    - parse(CharSequence, DateTimeFormatter)
- Métodos de format() lançam DateTimeException; métodos de parse() lançam DateTimeParseException.

Autoavaliação (Self Test)

1. Quais das seguintes são formas válidas de criar um objeto ZoneId?

A. ZoneId.ofHours(2);
B. ZoneId.of("2");
C. ZoneId.of("-1");
D. ZoneId.of("America/Canada");

---

2. Dado o código:

```java
ZoneOffset offset = ZoneOffset.of("Z");
System.out.println(
    offset.get(ChronoField.HOUR_OF_DAY)
);
```

Qual dos seguintes é o resultado da execução das linhas acima?

A. 0
B. 1
C. 12:00
D. Uma exceção é lançada

---

3. Dado:

```java
ZonedDateTime zdt =
    ZonedDateTime.of(2015, 02, 28, 5, 0, 0, 0,
                     ZoneId.of("+05:00"));
System.out.println(zdt.toLocalTime());
```

Assumindo um fuso horário local de +02:00, qual das seguintes é a saída?

A. 05:00
B. 17:00
C. 02:00
D. 03:00

---

4. Dado:

```java
ZonedDateTime zdt =
    ZonedDateTime.of(2015, 10, 4, 0, 0, 0, 0,
                     ZoneId.of("America/Asuncion"))
    .plus(Duration.ofHours(1));
System.out.println(zdt);
```

Assumindo que o horário de verão começa em 4 de outubro de 2015 às 00:00:00, qual é o resultado da execução das linhas acima?

A. 2015-10-04T00:00-03:00[America/Asuncion]
B. 2015-10-04T01:00-03:00[America/Asuncion]

C. 2015-10-04T02:00-03:00[America/Asuncion]
D. 2015-10-03T23:00-03:00[America/Asuncion]

---

5. Dado:

```java
ZonedDateTime zdt =
    ZonedDateTime.of(2015, 3, 22, 0, 0, 0, 0,
                     ZoneId.of("America/Asuncion"))
        .minus(Period.ofDays(1));
System.out.println(zdt);
```

Assumindo que o horário de verão termina em 22 de março de 2015 às 00:00:00, qual é o resultado?

A. 2015-03-21T01:00-03:00[America/Asuncion]
B. 2015-03-21T00:00-03:00[America/Asuncion]
C. 2015-03-20T23:00-03:00[America/Asuncion]
D. 2015-03-21T02:00-03:00[America/Asuncion]

---

6. Quais das seguintes afirmações são verdadeiras?

A. java.time.ZoneOffset é uma subclasse de java.time.ZoneId.
B. java.time.Instant pode ser obtido a partir de java.time.ZonedDateTime.
C. java.time.ZoneOffset consegue gerenciar DST.
D. java.time.OffsetDateTime representa um ponto no tempo no fuso horário UTC.

---

7. Dado:

```java
DateTimeFormatter formatter =
    DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
System.out.println(
    formatter
    .withLocale(Locale.ENGLISH)
    .format(LocalDateTime.of(2015, 5, 7, 16, 0))
);
```

Qual das seguintes é a saída?

A. 5/7/15 4:00 PM
B. 5/7/15
C. 4:00 PM
D. 4:00:00 PM

---

8. Dado:

```java
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("HH:mm:ss X");
OffsetDateTime odt =
    OffsetDateTime.parse("11:50:20 Z", formatter);
```

Qual das seguintes afirmações é verdadeira?

A. O padrão "HH:mm:ss X" é inválido.
B. Um OffsetDateTime é criado com sucesso.
C. Z é um deslocamento inválido.
D. Uma exceção é lançada em tempo de execução.